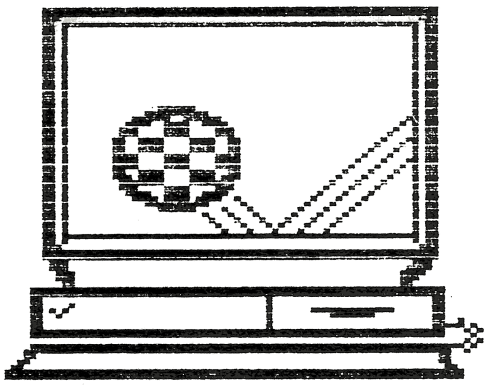


AMIGA MAIL™

June '86

WELCOME

Welcome aboard Mike Brenner! Mr. Brenner has joined Commodore as the new Vice-President of Software. Coming to Commodore from Zenith Data Systems, Mr. Brenner held the position of Director of Product Planning. We are confident that he will help see that the Amiga succeeds. Former Software Director Paul Goheen has resigned to pursue other interests. We all wish both Mike Brenner and Paul Goheen the best of luck!

SOFTWARE UPDATE:

Quelo has been providing programmers with software development tools since 1978. They continue to do so with the Quelo Assembler packages for the Amiga. With a wide assortment of host machines, software developers can find assistance in producing products for the Amiga. It also allows the Amiga to be used as a software development environment for external 68xxx targets.

For more information, contact Quelo, Inc. at 2464 33rd West, Suite 173, Seattle, WA 98199.

A-Talk by Felsina Software

Looking for an advanced communication and terminal program for your Amiga? Felsina Software recently announced the release of A-Talk. A-Talk includes integrated communication tools which work together to help you collect, control, and transmit data with your Amiga. A-Talk also supports KERMIT Wildcard sends/receives and strips executable objects of extra data sent by Xmodem programs.

A-Talk for the Amiga lists for $49.95. For more information contact Felsina Software at 3175 S. Hoover St., Suite #275, Los Angeles, CA 90007 or call (213) 747-8498.

SUPPORT SERVICES

Due to finite resources, we are forced to restrict some of our developer support services. However, we are willing to provide a number of materials to any Amiga developer.

AMIGA DEVELOPER'S KIT:

Contents: DOS User's Manual, DOS Technical Reference Manual, DOS Developer's Manual, Intuition Manual, Hardware Manual, ROM Kernal Manual, Lattice 'C' Compiler and manual, 68000 Macro Assembler, and the IBM cross-development environment.
Developer's kit price: $450.00

AMIGA DOCUMENTATION:

contents: the six Amiga manuals from the kit.
Documentation price: $99.95

IFF DOCUMENTATION & DISK:

IFF price: $20.00

HARDWARE SCHEMATICS:

contents: schematics and expansion specifications.
Schematics price: $20.00

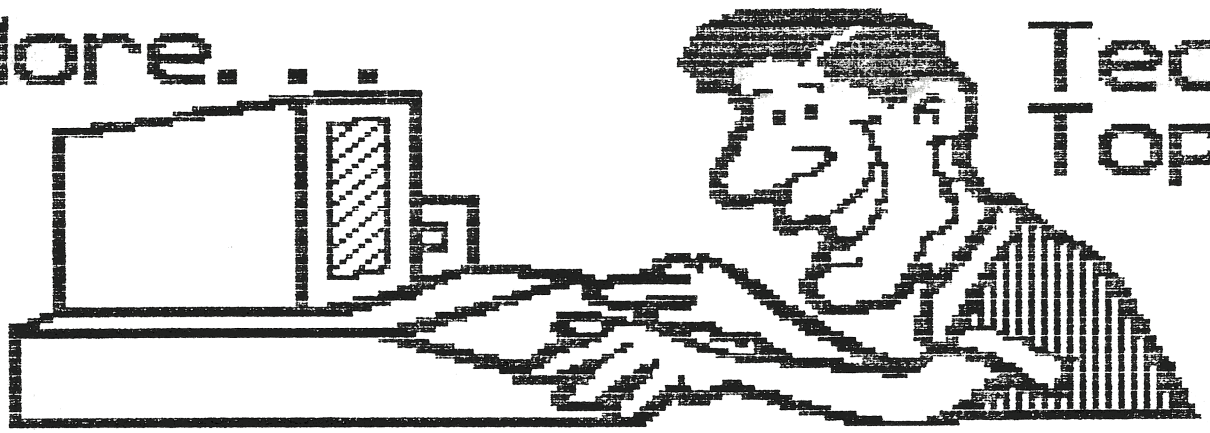
To order, send a check payable to COMMODORE BUSINESS MACHINES, INC. to:

CBM
ATTN: Bernadette Loughery
1200 Wilson Drive
West Chester, PA 19380

Be sure to include your name and address with a note requesting the particular item.

These materials are provided as is. In the past, we have provided free upgrades; however, this is not a policy . Given the scale of the upgrades, generally it is not financially feasible to send the materials out free of charge.

More... Tech Topics



1.2 Bug Reports

Any developer currently working with Beta 1.2 is encouraged to post any bugs they encounter to Commodore-Amiga. The bug reports may be sent to :

Commodore- Amiga, Inc.
Attn: Nancy Rains
983 University Avenue
Building D
Los Gatos, CA 95030

Syncing SuperBitMap for Printing & Saving

```
/*  Code fragment
 *   window is a SuperBitMap window
 *   rp is window->RPort
 *   Functions from graphics.library
 *   Pass &dRastPort when dumping
 */

 struct RastPort dRastPort; /*dummy*/

/* Lock the Layer
 * Update the SuperBitMap
 * Copy rastport to dummy
 * Insert ptr to our SuperBitMap
 * NULL the Layer ptr
 * Unlock the Layer
 */

 LockLayerRom(rp->Layer);
 SyncSBitMap(rp->Layer);
 dRastPort = *rp;
 dRastPort.BitMap =
     rp->Layer->SuperBitMap;
 dRastPort.Layer = NULL;
 UnlockLayerRom(rp->Layer);

 /* Carolyn, Jim, Barry - CBM & Amiga */
```

REMINDER!

Did you return the survey included in the April edition of Amiga Mail? If not, please send it as soon as possible. We are currently updating our Amiga Developer Database. The information included on the survey is crucial in determining status for upgrades, mailouts, and general Amiga software information. If you have not received an application or have misplaced yours, please call (215) 431-9180. We will send another out to you.

WANTED! YOUR QUESTIONS

We want to hear from you! If you have any technical, developmental, or marketing questions, SEND 'EM IN! Our technical support group will do their darndest to answer your questions. So don't be a bump-on-a-log, get those questions to Becky Cotton at our Commodore address so she can stick 'em in our newsletter.

Software Department
Commodore Business Machines
1200 Wilson Drive
West Chester, PA
19380

Here's a program you will need when making printer drivers. It was accidently left out of the Rom Kernal Manual, Vol II.
(By the way, it also serves as an example of using the timer device in assembler)

andy finkel

```
        TTL     '$Header$'
********************************************************************************
*                                                                              *
*   Copyright 1985, Commodore-Amiga Inc.   All rights reserved.                *
*   No part of this program may be reproduced, transmitted,                    *
*   transcribed, stored in retrieval system, or translated into                *
*   any language or computer language, in any form or by any                   *
*   means, electronic, mechanical, magnetic, optical, chemical,                *
*   manual or otherwise, without the prior written permission of               *
*   Commodore-Amiga Incorporated, 983 University Ave. Building #D,             *
*   Los Gatos, California, 95030                                               *
*                                                                              *
********************************************************************************
*
*       wait
*

        SECTION         printer

*------ Included Files ---------------------------------------------

        INCLUDE         "exec/types.i"
        INCLUDE         "exec/ports.i"
        INCLUDE         "exec/devices.i"
        INCLUDE         "exec/io.i"

        INCLUDE         "devices/timer.i"

        INCLUDE         "../printer/macros.i"
        INCLUDE         "../printer/prtbase.i"


*------ Imported Names ---------------------------------------------

*------ Imported Functions -----------------------------------------

        XREF_EXE        Forbid
        XREF_EXE        Permit
        XREF_EXE        WaitIO
        XREF            _SysBase

        XREF            _PD


*------ Exported Functions -----------------------------------------

        XDEF            _PWait


*------ printer.device/PWait ---------------------------------------
*
*   NAME
*       PWait - wait for a time
*
*   SYNOPSIS
*       PWait(seconds, microseconds);
*
*   FUNCTION
*       PWait uses the timer device to wait after writes are complete
*
```

```
*-------------------------------------------------------------------------
_PWait:
                MOVEM.L A4/A6,-(A7)
                MOVE.L  _PD,A4
                MOVE.L  pd_PBothReady(A4),A0
                JSR     (A0)
                TST.L   D0
                BNE.S   error

                LEA     pd_TIOR(A4),A1
                MOVE.W  #TR_ADDREQUEST,IO_COMMAND(A1)
                MOVE.L  12(A7),IOTV_TIME+TV_SECS(A1)
                MOVE.L  16(A7),IOTV_TIME+TV_MICRO(A1)
                CLR.B   IO_FLAGS(A1)
                MOVE.L  IO_DEVICE(A1),A6
                JSR     DEV_BEGINIO(A6)
                LINKEXE Forbid
                LEA     pd_TIOR(A4),A1
                LINKEXE WaitIO
                LINKEXE Permit
                MOVEQ   #0,D0
                TST.L   D0
error:
                MOVEM.L (A7)+,A4/A6
                RTS

                END
```

MULTIPLE-PROCESSES
MULTIPLE-PROCESSES by Rob Peck
MULTIPLE-PROCESSES
MULTIPLE-PROCESSES
MULTIPLE-PROCESSES

One of the questions that is often asked about the Amiga is:

HOW DO I USE THE MULTI-TASKING CAPABILITIES?

There are actually two different levels of multi-tasking that are possible on the Amiga, and which of these that you use depends on the nature of the subtask you wish to perform.

- If your code is largely self-contained, and does no I/O and uses no disk-resident library code, you can probably spawn a task, as is demonstrated in the task-spawning message.

- If your code needs to do I/O, (or uses AmigaDOS functions in ANY way), it will have to be spawned as a process rather than as a task. MULTIPLE-PROCESSING is shown here.

WHAT IS COVERED HERE?

There are two programs contained in this message:

- littleproc.c - a demonstration routine that is spawned by another process. The normal startup code with which it is linked automatically waits for a workbench-startup message before it gets going. Using the same message port that was provided when the process was initiated, it again goes to sleep waiting for a message that contains specific information... in this case, the parameters that the master program is using, namely its stdout and stderr filehandles. Thus, this spawned process can be made to output to the same window from which the originating process was begun.

- proctest.c - a demo program that loads and starts littleproc, and unloads its code and data when it finishes.

A process is a superset of a task, and the various AmigaDOS routines require that a process control block and its associated information be available in order to run. Thus this code is provided to allow a user who requires a process rather than a task to have an example on which to build.

The example was tested from the CLI, and compiled under Amiga C, (Lattice 3.03).

Link information:

```
; "process.with"

FROM lib:Astartup.obj process.o
TO process
```

```
LIBRARY lib:amiga.lib

; "littleproc.with"

FROM lib:Astartup.obj littleproc.o
TO littleproc
LIBRARY lib:amiga.lib
```

DISCLAIMER:

This program is provided as a service to the programmer community to demonstrate one or more features of the Amiga personal computer. These code samples may be freely used for commercial or noncommercial purposes.

Commodore Electronics, Ltd ("Commodore") makes no warranties, either expressed or implied, with respect to the program described herein, its quality, performance, merchantability, or fitness for any particular purpose. This program is provided "as is" and the entire risk as to its quality and performance is with the user. Should the program prove defective following its purchase, the user (and not the creator of the program, Commodore, their distributors or their retailers) assumes the entire cost of all consequent damages. In no event will Commodore be liable for direct, indirect, incidental or consequential damages resulting from any defect in the program even if it has been advised of the possibility of such damages. Some laws do not allow the exclusion or limitation of implied warranties or liabilities for incidental or consequential damages, so the above limitation or exclusion may not apply.

Program dependencies:

Use c-devel:examples/makesimple on proctest.c and littleproc.c BUT modify makesimple to use:

Astartup.obj and amiga.lib only instead of Lstartup.obj and lc.lib+amiga.lib.

AND specify "-v" option (delete stack checking) for lc2. This eliminates any need to link with lc.lib or Lstartup.obj for this particular example.

Note: I haven't tried to make this compatible with the Lattice startup code.... the only point of incompatibility is my own use of stdout/stderr. Lattice defines them differently. AmigaDOS stdout and stderr are BPTR's to an AmigaDOS data structure. When using amiga.lib to provide "printf" at link time, printf internally uses the AmigaDOS version of stdout. If you link with lc.lib specified first, then it uses its own interpretation of stdout, so they will not be compatible with the values received from

the Open() function ("O"pen() is AmigaDOS, "o"pen is Lattice).

Lattice defines stdout/stderr as the address of an i/o block. If the main process is compiled under Lattice and the slave process is also compiled under Lattice, then the values passed for stdout and stderr will be compatible. If not, then there is probably something a person could do to make them compatible.

Perhaps some other person would care to investigate this and provide a translation method between the two. (Its a problem when one tries to mix two different manufacturer's idea of underlying code support.)

The purpose of the program is to allow you to see how a slave process can be initialized and started. Please ignore the stdout/stderr difficulties, and utilize whatever methods you wish for I/O.

```
/* proctest.c ************************************************** */

/* author:  Rob Peck   3/14/86                                   */
/* system software version: V1.1                                 */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/libraries.h"
#include "exec/ports.h"
#include "exec/interrupts.h"
#include "exec/io.h"
#include "exec/memory.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

#include "workbench/startup.h"

#define PRIORITY 0
#define STACKSIZE 5000

extern struct Message *GetMsg();
extern int LoadSeg();
extern struct MsgPort *CreateProc();
extern struct MsgPort *CreatePort();

struct MyMess {
   struct Message mm_Message;
   int            mm_OutPointer;
   int            mm_ErrPointer;
};

extern int stdout;
```

```
extern int stderr;

main()
{
        struct Message *reply;
        struct Process *myprocess;

        /* Message that we send to the process to wake it up */

        struct WBStartup *msg;

        /* Message to contain my own parameters to pass on to spawned
         * process, sample only.  Just to prove that we correctly
         * create a process, we are giving it something other than nil:
         * as its stdout and stderr... in fact, giving it OUR values
         * so we can share the same output window.
         */
        struct MyMess *parms;

        /* Because main() is itself started as a process, it automatically
         * has a message port allocated for itself.  Located at
         *  &((struct Process *)FindTask(0))->pr_MsgPort
         */

        int littleSeg;

        /* Actually littleSeg is a BPTR, but the int declaration
         * keeps the compiler happy and we don't use the
         * value ourselves anyhow... just pass it on.
         */

        char *startname, *parmname;

        struct MsgPort *mainmp;   /* pointer to main's msg port */
        struct MsgPort *littleProc; /* pointer to spawned proc's msg port */

/* Provide names for the messages we are passing so we can check the returned
 * messages at the message ports.... that is if we choose to do so.
 */
        startname = "startermessage";
        parmname  = "parameterpass";

/* LOAD THE PROGRAM TO BE STARTED FROM MAIN ****************************** */

        littleSeg = LoadSeg("littleproc");
        if(littleSeg == 0)
        {
                printf("\nlittleproc not found");
                exit(999);
        }

/* CREATE A PROCESS FOR THIS OTHER PROGRAM ******************************* */

        littleProc = CreateProc("littleguy",PRIORITY, littleSeg, STACKSIZE);
        if( littleProc == 0 )
        {
```

```
                printf("\Couldn't create the process");
                UnLoadSeg(littleSeg);
                exit( 1000 );
        }

/* ********************************************************************** */
/* Create a msgport */

        myprocess = (struct Process *)FindTask(0);

        mainmp = CreatePort(0,0);     /* should error check here */

/* ********************************************************************** */
/* THE FOLLOWING CODE BLOCK STARTS THE PROCESS RUNNING,
                                        AS THOUGH CALLED FROM WBENCH */

/*      In fact, because we created the process the way that is shown
        here, if you use the standard startup code, the program must
        be started as though called from Workbench.  It is now waiting for
        a startup message.

        (There is, in fact, another way to call a loaded program's code,
        but it does not entail starting another process.  Rather it
        uses a direct call (as a subroutine) to the loaded code.  The
        other program runs on your own stack, so your program must
        have sufficient stack to handle both of you.  It also runs
        under your own process, so your own program does not get
        control until that other program has completed.  The program
        return()'s or exit()'s to you, providing the appropriate
        returncode.  This run-loaded-subroutine topic is covered in
        a separate code sample.)

********************************************************************** */

        /* This message block is a wakeup call to the process we created. */
        msg = (struct WBStartup *)AllocMem(sizeof(struct WBStartup),
                MEMF_CLEAR);
        if(msg)
        {
                /* Preset the necessary arguments for message passing */

                msg->sm_Message.mn_ReplyPort = mainmp;
                msg->sm_Message.mn_Length = sizeof(struct WBStartup);
                msg->sm_Message.mn_Node.ln_Name = startname;

                /* Passing no workbench arguments to this process;
                 * we are not WBench.  Of course, if we want to pass
                 * workbench-style arguments this way, we can.
                 */

                msg->sm_ArgList = NULL;

                /* If the process is being opened without a ToolWindow
                 * (Workbench sets this up) as a parent, slave will simply
                 * go on to do its own main()  ... as shown in Astartup.asm
                 */
```

```
        msg->sm_ToolWindow = NULL;

        /* Send the startup message */

        PutMsg(littleProc,msg);
}
else
{
        printf("\nCouldnt allocate mem for WBStartup!\n");
        goto aarrgghh;  /* Oh no, a "goto" ! */
}
/* ********************************************************** */
/* Just a sample message, still using the same message and
 * reply ports
 *
 * Littleproc is a cooperating process...it KNOWS it must wait
 * until a message arrives at its port, containing the parameters
 * it should use for output.
 *
 * The startup message is handled by the standard startup code.
 * This parameter message is handled by the program code itself.
 * The startup message is returned to the replyport by the startup
 * code, after the program code exits or returns.
 */

parms = (struct MyMess *)AllocMem(sizeof(struct MyMess),MEMF_CLEAR);
if(parms)
{
        parms->mm_Message.mn_ReplyPort = mainmp;
        parms->mm_Message.mn_Length = sizeof(struct mymess);
        parms->mm_Message.mn_Node.ln_Name = parmname;

        /* NOTE THAT THESE ARE THE AStartup.asm stdout and stderr;
         * the example works only if both master and slave are
         * compiled and linked with the same startup code. */

        parms->mm_OutPointer = (int)stdout;
        parms->mm_ErrPointer = (int)stderr;
        /* send him our parameters */

        PutMsg(littleProc,parms);

        /* wait for the reply from parameter pass. */

        WaitPort(mainmp);

        reply = GetMsg(mainmp);

        /* Message node name should contain the address of the
         * string "parms" if error checking was included.
         *
         *
         * User should probably allocate separate ports for
         * parameter passing different from the main port
         * automatically allocated by the system when a
```

```
                 * process is initiated.  It would alleviate
                 * some of the checking that is appropriate to do
                 * when multiple kinds of messages arrive at the same port.
                 *
                 *
                 *       NOW MAIN CAN GO ON AND DO SOMETHING USEFUL,
                 *       LATER CAN COME BACK AND SEE IF SPAWNED PROCESS
                 *       HAS COMPLETED AND IS READY TO BE UNLOADED.
                 *
                 *
                 * Wait for the return of the wbstartup message before
                 * main itself is allowed to exit.
                 */

                WaitPort(mainmp);

                reply = GetMsg(mainmp);

                /* Message node name should be
                 * address of "startermessage"  */

                /* NOTE: there should be checking here to see if the message
                 * received at this port was the string, or the wakeup call.
                 * This sample code only assumes that the string is received
                 * and replied first, then the wakeup call message is returned
                 * as the little task is exiting.
                 */

                UnLoadSeg(littleSeg);
                DeletePort(mainmp);

                printf("\nSlave exited; Master unloaded its code and data\n");
        }
        else
        {
                printf("\nCouldn't allocate memory for parameter message\n");
        }
    aarrgghh:
                /* arrive here on good or bad exit */

        if(parms) { FreeMem( parms, sizeof(struct MyMess));    }
        if(msg)   { FreeMem( msg,   sizeof(struct WBStartup)); }

}       /* end of main */



/* littleproc.c ********************************************************* */

/* Sample slave code for create process test */

/* author: Rob Peck   3/4/86  */

/* system software version: V1.1                */
```

```
#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/libraries.h"
#include "exec/ports.h"
#include "exec/interrupts.h"
#include "exec/io.h"

#include "libraries/dos.h"
#include "libraries/dosextens.h"

#include "workbench/startup.h"

/* these are going to be supplied to the slave by the starter */
/* they are actually defined in the startup code (Astartup.asm) */

extern int stdout;
extern int stderr;

struct MyMess {
    struct Message mm_Message;
    int mm_OutPointer;
    int mm_ErrPointer;
};

extern struct Message *GetMsg();
extern struct Task *FindTask();
extern struct FileHandle *Open();

main()
{
        struct MyMess *msg;
        struct MsgPort *myport;
        struct Process *myprocess;

        struct FileHandle *myOwnOutput;

        myprocess = (struct Process *)FindTask(0);

        myport = &myprocess->pr_MsgPort;

        /* Wait for starter to post a message.  Special sample message
         * has his stderr, stdout so we can both post stuff to the
         * same CLI window as he started from */

        WaitPort(myport);

        msg = (struct MyMess *)GetMsg(myport);


        stdout = msg->mm_OutPointer;

        /* Use printf to prove that it is really a process...
         * a simple task cannot do this without crashing! */

        printf("\nHere I am, that slave process you started!!!");
```

```
        printf("\nNow going to open MY OWN window.\n");

        /* NOW DO SOMETHING USEFUL... DO WHATEVER THE PROCESS WAS DESIGNED
         * TO ACCOMPLISH.
         */

        myOwnOutput = Open("CON:10/10/320/150/SlaveProcess",MODE_NEWFILE);
        if(myOwnOutput == 0)
        {
                ReplyMsg(msg);  /* tell main I'm done */
                exit(0);        /* can't return an error code anyhow */
        }
        else
        {
                stdout = (int)myOwnOutput;
                /* reset my output file handle */
                printf("See, I can do AmigaDOS!");
                Delay(250);     /* 250/50 = 5 seconds */
                stdout = msg->mm_OutPointer;
                Close(myOwnOutput);
                ReplyMsg(msg);
        }

        /* Now simply fall off the end of the world,
         * returns to the startup code, and should exit cleanly */

}

/* ****************************************************************

As a final note, one could have created and started another process by
using the Execute command of AmigaDOS:

        success = Execute("someprogram",0,0);

but I went through this exercise to show interprocess communication
setup and message passing.  I hope that the sample code provides some
insite into the multi-processing capabilities of AmigaDOS.

**************************************************************** */
```